INDEPENDENT NET TASK IDENTIFICATION FOR EFFICIENT PARTITION AND DISTRIBUTION

DESCRIPTION

RELATED APPLICATION

Sub A1

The present application is related to U.S. Patent Application No. 09/__________ (Attorney Docket No. YOR9-2000-0464-US1) entitled “MACHINE CUT TASK IDENTIFICATION FOR EFFICIENT PARTITION AND DISTRIBUTION” to Rajan et al.; U.S. Patent Application No. 09/__________ (Attorney Docket No. YOR9-2000-0465-US1) entitled “NET ZEROING FOR EFFICIENT PARTITION AND DISTRIBUTION” to Roth et al.; and U.S. Patent Application No. 09/__________ (Attorney Docket No. YOR9-2000-0466-US1) entitled “DOMINANT EDGE IDENTIFICATION FOR EFFICIENT PARTITION AND DISTRIBUTION” to Wegman et al. all filed coincident herewith and assigned to the assignee of the present invention.

BACKGROUND OF THE INVENTION

Field of the Invention

The present invention generally relates to distributed processing and more particularly, the present invention relates to efficiently assigning tasks across multiple computers for distributed processing.

Background Description

Any large, multifaceted project, such as a complex computer program, may be segmented into multiple smaller manageable tasks. The tasks then may be distributed amongst a group of individuals for independent completion, e.g., an engineering design project, distributed processing or, the layout of a complex electrical circuit such as a microprocessor. Ideally, the tasks are matched with the skills of the assigned individual and each task is completed with the same effort level as every other task. However, with such an ideal matched task assignment, intertask communication can become a bottleneck to project execution and completion. Thus, to minimize this potential bottleneck, it is important to cluster together individual tasks having the highest level of communication with each other. So, for example, in distributing eight equivalent tasks to pairs of individuals at four locations, (e.g., eight design engineers in four rooms) optimally, pairs of objects or tasks with the highest communication rate with each other are assigned to individual pairs at each of the four locations.

Many state of the art computer applications are, by nature, distributed applications. End-users sit at desktop workstations or employ palmtop information appliances on the run, while the data they need to access resides on distant data servers, perhaps separated from these end-users by a number of network tiers. Transaction processing applications manipulate data spread across multiple servers. Scheduling applications are run on a number of machines that are spread across the companies of a supply chain, etc.

When a large computer program is partitioned or segmented into modular components and the segmented components are distributed over two or more machines, for the above mentioned reasons, component placement can have a significant impact on program performance. Therefore, efficiently managing distributed programs is a major challenge, especially when components are distributed over a network of remotely connected computers. Further, existing distributed processing management software is based on the assumption that the program installer can best decide how to partition the program and where to assign various program components. However, experience has shown that programmers often do a poor job of partitioning and component assignment.

So, a fundamental problem facing distributed application developers is application partitioning and component or object placement. Since communication cost may be the dominant factor constraining the performance of a distributed program, minimizing inter-system communication is one segmentation and placement objective. Especially when placement involves three or more machines, prior art placement solutions can quickly become unusable, i.e., what is known as NP-hard. Consequently, for technologies such as large application frameworks and code generators that are prevalent in object-oriented programming, programmers currently have little hope of determining effective object placement without some form of automated assistance. En masse inheritance from towering class hierarchies, and generation of expansive object structures leaves programmers with little chance of success in deciding on effective partitioning. This is particularly true since current placement decisions are based solely on the classes that are written to specialize the framework or to augment the generated application.

Furthermore, factors such as fine object granularity, the dynamic nature of object-based systems, object caching, object replication, ubiquitous availability of surrogate system objects on every machine, the use of factory and command patterns, etc., all make partitioning in an object-oriented domain even more difficult. In particular, for conventional graph-based approaches to partitioning distributed applications, fine-grained object structuring leads to enormous graphs that may render these partitioning approaches impractical.

Finally, although there has been significant progress in developing middleware and in providing mechanisms that permit objects to inter-operate across language and machine boundaries, there continues to be little to help programmers decide object-system placement. Using state of the art management systems, it is relatively straightforward for objects on one machine to invoke methods on objects on another machine as part of a distributed application. However, these state of the art systems provide no help in determining which objects should be placed on which machine in order to achieve acceptable performance. Consequently, the initial performance of distributed object applications often is terribly disappointing. Improving on this initial placement performance is a difficult and time-consuming task.

Accordingly, there is a need for a way of automatically determining the optimal program segmentation and placement of distributed processing components to minimize communication between participating distributed processing machines.

SUMMARY OF THE INVENTION

It is therefore a purpose of the present invention to improve distributed processing performance;

It is another purpose of the present invention to minimize communication between distributed processing machines;

It is yet another purpose of the invention to improve object placement in distributed processing applications;

It is yet another purpose of the invention to determine automatically how objects should best be distributed in distributed processing applications;

it is yet another purpose of the invention to minimize communication between objects distributed amongst multiple computers in distributed processing applications.

The present invention is a task management system, method and computer program product for determining optimal placement of task components on multiple machines for task execution, particularly for placing program components on multiple computers for distributed processing. First, a communication graph is generated representative of the computer program with each program unit (e.g., an object) represented as a node in the graph. Nodes are connected to other nodes by edges representative of communication between connected nodes. A weight is applied to each edge, the weight being a measure of the level of communication between the connected edges. Terminal nodes representative of the multiple computers are attached to the communication graph. Then, the communication graph is divided into independent nets and a min cut is found for each independent net. The min cut for the communication graph is the combination of the min cuts for all of the independent nets. Finally, program components which may be a single program unit or an aggregate of units are placed on computers according to the communication min cut.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, aspects and advantages will be better understood from the following detailed preferred embodiment description with reference to the drawings, in which:

Figure 1 is an example of a flow diagram of the preferred embodiment of the present invention wherein a program is segmented, initially, and initial segments are distributed to and executed on multiple computers;

Figures 2A-C show an example of a communication graph;

Figure 3 is a flow diagram of the optimization steps for determining an optimum distribution of program components;

Figure 4 is an example of a communication net that includes two independent nets;

Figure 5 shows the steps of identifying independent nets;

Figures 6A-B show a flow diagram showing a detailed example of the steps in finding the min cut for each independent net.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS OF THE INVENTION

As referred to herein, a communication graph is a graphical representation of a multifaceted task such as a computer program. Each facet of the task is an independent task or object that is represented as a node and communication between tasks or nodes is represented by a line (referred to as an edge) between respective communicating nodes. Participating individuals (individuals receiving and executing distributed tasks) are referred to as terminal nodes or machine nodes. A net is a collection of nodes connected together by edges. Two nets are independent if none of the non-terminal nodes of one net shares an edge with a non-terminal node of the other. Thus, for example, a communication graph of a computer program might include a node for each program object and edges would be between communicating objects, with edges not being included between objects not communicating with each other. In addition, a weight indicative of the level of communication between the nodes may be assigned to each edge. Graphical representation and modeling of computer programs is well known in the art.

Referring now to the drawings, and more particularly, Figure 1 is an example of a flow diagram 100 of the preferred embodiment of the present invention wherein a program is segmented, initially, and initial segments are distributed to and executed on multiple computers. First, in step 102 the communication patterns of a program are analyzed to form a communication graph. Then, in step 104, the traces of the communication graph are analyzed, and an initial partition is determined. In step 106, the partition is optimized for minimum interpartition communication. In step 108, the individual objects are distributed amongst participants for execution according to the optimize partition of step 106.

A component refers to an independent unit of a running program that may be assigned to any participating individual, e.g., computers executing objects of a distributed program. Thus, a component may refer to an instance of an object in an object oriented program, a unit of a program such as Java Bean or Enterprise Java Bean or, a larger collection of program units or components that may be clustered together intentionally and placed on a single participating machine. Further, a program is segmented or partitioned into segments that each may be a single component or a collection of components. After segmenting, analyzing the segments and assigning each of segments or components to one of the multiple participating machines or computers according to the present invention, the final machine assignment is the optimal assignment.

Thus, a typical communication graph includes multiple nodes representative of components with weighted edges between communicating nodes. Determining communication between components during program execution may be done using a typical well known tool available for such determination. Appropriate communication determination tools include, for example, Jinsight, for Java applications that run on a single JVM, the Object Level Tracing (OLT) tool, for WebSphere applications or, the monitoring tool in Visual Age Generator.

Figures 2A-C show an example of a communication graph of a net 110 that includes multiple nodes 112, 114, 116, 118, 120 and 122. Each node 112, 114, 116, 118, 120 and 122 represents a program component connected to communication edges 124, 126, 128, 130, 132, 134, 136 and 138 to form the net 110. Adjacent nodes are nodes that share a common edge, e.g., nodes 114 and 122 share edge 126. Each edge 124, 126, 128, 130, 132, 134, 136 and 138 has been assigned a weight proportional to, for example, the number of messages between the adjacent components.

In Figure 2B, Machine nodes 140, 142 and 144 representative of each participating machine (three in this example) are shown connected by edges 146, 148, 150. Initially, a node 112, 114, 116, 118, 120 and 122 may be placed on a machine 140, 142, 144 by adding an edge 146, 148, 150 with infinite weight (indicating constant communication) between the node and the machine. Typically, initial assignment places nodes with specific functions (e.g., database management) on a machine suited for that function. After the initial placement assigning some nodes 112, 114 and 122 to machines 140, 142, 144, other nodes 116, 118, 120 are assigned to machines 140, 142, 144, if they communicate heavily with a node 112, 114, 122 already assigned to that machine 140, 142, 144. Additional assignment is effected by selectively collapsing edges, combining the nodes on either end of the collapsed edge and re-assigning edges that were attached to one of the two former adjacent nodes to the combined node. When assignment is complete, all of the nodes 112, 114, 116, 118, 120 and 122 will have been placed on one of the machines at terminal nodes 140, 142, 144 and the final communication graph may be represented as terminal nodes 140, 142, 144 connected together by communication edges.

For this subsequent assignment, the graph is segmented by cutting edges and assigning nodes to machines as represented by 152, 154 and 156 in Figure 2C to achieve what is known in the art as a minimal cut set or min cut set. A cut set is a set of edges that, if removed, eliminate every path between a pair of terminal nodes (machine nodes) in the graph. A min cut set is a cut set wherein the sum of the weights of the cut set edges is minimum. While there may be more than one min cut set, the sum is identical for all min cut sets. A min cut may be represented as a line intersecting the edges of a min cut set. So, in the example of Fig. 2C, the sum of the weights of edges 124, 126, 128, 132 and 138 is 2090, which is cost of the cut and is representative of the total number of messages that would be sent between machines at terminal nodes 140, 142, 144 with this particular placement. The min cut identifies the optimum component placement with respect to component communication. While selecting a min cut set may be relatively easy for this simple example, it is known to increase in difficulty exponentially with the number of nodes in the graph.

Figure 3 is a flow diagram 160 of the optimization steps for determining an optimum distribution of program components to individual participating computers according to a preferred embodiment of the present invention. First, in step 162, an initial communication graph is generated for the program. Then, in step 164 machine nodes are added to the communication graph. As noted above, certain types of components are designated, naturally, for specific host machine types, e.g., graphics components are designated for clients with graphics capability or, server components are designated for a data base server. After assigning these host specific components to host nodes, in step 168 independent nets are identified and the communication graph is partitioned into the identified independent nets. In step 170 a min cut is found for each independent net, the min cuts for all of the independent nets being the min cut for the whole communication graph.

Figure 4 is an example of a communication net 180 that includes two independent nets 182, 184. Independent net 182 includes all of the nodes represented by black dots and all of the corresponding connecting edges. Independent net 184 includes all of the nodes represented by white dots and all of the corresponding connecting edges. Both independent nets 182, 184 include terminal nodes 186, 188.

Figure 5 show the steps of identifying independent nets of step 168 of Figure 3. The communication graph is partitioned into subgraphs that are independent nets by repeatedly picking a "seed node" from nodes remaining in the communication graph and branching out until terminal nodes have been reached along all paths. As noted above, an independent net is a subset of components that do not communicate with any other subset of components. Thus, it should be noted that the communication graph may or may not be divisible into independent nets. However, independent net identification begins with initialization in step 1682 by marking all the unallocated nodes (nodes that have not been assigned to a terminal node) as being unvisited. Also, during this initialization step 1682, a set of independent nets (e.g., a list of subgraphs) is created and initialized as empty or null set. Since all unallocated nodes have been initially marked unvisited,

unvisited nodes are found in the first instance of step 1684. So, in step 1686 an unvisited node (u) is selected as a starting point to identify an independent net and the independent net is entered into the set of nets. At this point, a new traversal stack is initially created with selected node u being the sole first entry and u is marked as visited. The traversal stack maintains a list of the outer visited or perimeter non-terminal nodes of the independent net and is empty when the independent net has been fully identified.

So, except for the trivial case where the independent net is a single node net, when the traversal stack is first checked for any independent net in step 1688, the traversal stack is not empty. Until the traversal stack is found to be empty in step 1688, one visited node is popped from the stack in step 1690. All of the immediate neighbors (adjacent nodes) to the popped node are pushed onto the traversal stack, marked as visited and, thereby, perimeter nodes included in the independent net being formed. Correspondingly, all of the edges between the popped node and its newly added neighbors also are included in the independent net. Repeatedly, the traversal stack is checked in step 1688, perimeter nodes are popped from the stack and any adjacent nodes are added to the stack as perimeter nodes in step 1690. This is repeated until the traversal stack is found to be empty in step 1688 and gathering of non-terminal nodes for this independent net is complete. So, in step 1692 any terminal node incident to an edge already in the net are added. After step 1692, returning to step 1684 if any unvisited nodes remain, then additional independent nets remain and independent net identification continues in step 1686. However, if no unvisited nodes are found in step 1684, then all independent nets have been identified, only terminal nodes are perimeter nodes and the independent net identification step 168 is complete in step 1694. The output of step 168 is a set of independent nets in the communication graph. If the output is only a single independent net, then no simplification of the communication graph occurred.

Figures 6A-B are a flow diagram showing a detailed example of the steps in finding the min cut in step 170 for each independent net to achieve an optimum program component distribution to individual participating computers according to the preferred embodiment of the present invention. In this example, a multiway minimum cut of the communication graph is identified by sequencing graph reduction methods and independent net formation. In this way, a complex communication graph can be reduced quickly, preferably using linear complexity heuristics or methods, to a small manageable size to allow finding the multiway minimum cut very quickly using any well known method, i.e:, methods that have higher computational complexity but, would otherwise be totally impractical on the full communication graph absent the reduction of the present invention.

So, after isolating independent nets in step 168, each independent net being listed, for example, as such in a subgraph list, the subgraph list is passed to step 170. First, in step 1702 the subgraph list is checked to determine if the min cut has been found for all of the subgraphs, i.e., the subgraph list is empty. It is understood that as the min cut step 170 progresses, some subgraphs may be further divided into smaller subgraphs and those smaller subgraphs included in the subgraph list. However, in the first pass through step 1702 the min cut will not have been found for at least one subgraph. So in step 1704, one independent net or subgraph is selected and removed from the subgraph list. This selection may be arbitrary or based on available subgraph internal state information. Then, in step 1706 the selected independent net is checked using a linear complexity method (e.g., M. Padberg and G. Rinaldi "An Efficient Algorithm for the Minimum Capacity Cut Problem", *Math. Prog.* 47, 1990, pp. 19-36) to determine if it can be reduced in step 1708.

Preferably, however, the linear complexity methods employed in step 1706 include U.S. Patent Application No. 09/__________ (Attorney Docket No. YOR9-2000-0464-US1) entitled "MACHINE CUT TASK IDENTIFICATION FOR EFFICIENT PARTITION AND DISTRIBUTION" (Machine Cut) to Rajan et al.; U.S. Patent Application No. 09/__________ (Attorney Docket No. YOR9-2000-0465-US1) entitled "NET ZEROING FOR EFFICIENT PARTITION AND DISTRIBUTION" (Zeroing) to Roth et al.; and U.S. Patent Application No. 09/__________ (Attorney Docket No. YOR9-2000-0466-US1) entitled "DOMINANT EDGE IDENTIFICATION FOR EFFICIENT PARTITION AND DISTRIBUTION" (Dominant Edge) to Wegman et al., filed coincident herewith, assigned to the assignee of the present invention and incorporated herein by reference. Most preferably, in step 1708 the Dominant Edge method is used first, followed by Zeroing and then, by the Machine Cut method. This reduction may involve collapsing edges (Dominant Cut and Machine Cut) or reducing edge weights (Zeroing) and then collapsing edges. To reach a solution more quickly, on each pass through step 1706, only nodes and edges of a subgraph that were adjacent to areas reduced previously in step 1708 are rechecked.

After reducing the subgraph using one or more linear complexity methods, in step 1710 in Figure 6B, the subgraph is checked to determine if, as a result of the reduction, the subgraph is complete. The subgraph is complete either, if the min cut has been found or, if the reduction of step 1708 has partitioned the graph into independent nets. So, if the subgraph is complete, returning to step 1702, the min cut step 170 continues. Otherwise, returning to step 1706 the reduced subgraph is again checked to determine if it can be reduced using a linear complexity method, e.g., using the Dominant Cut and/or Machine cut after reducing edge weights.

~~If it is determined in step 1706 that the subgraph cannot be reduced using a linear complexity method, then, in step 1712 the subgraph is checked to determine if it may be further divided into independent nets and, if so, it is divided into independent subgraphs in step 1714 as described above with reference to Figure 5. Initially, as a result of the original independent net identification of step 168, the subgraphs cannot be further divided into independent nets. However, as subgraphs are simplified, e.g., though reduction using a linear complexity method in step 1708, the reduced subgraph may be further divided into independent subgraphs. So, if in step 1712 it is determined that the subgraph includes two or more independent nets, then in step 1714, the subgraph is partitioned into independent nets which are added to the subgraph list and the patitioned subgraph is complete. Thus, in step 1710, when the subgraph is checked, it is determined to be complete and, returning to step 1702, the min cut step 170 continues.~~

Sub A7

In step 1712, if it is determined that the subgraph is not further partitionable, the subgraph is checked in step 1716 whether it may be reduced using a more complex reduction method than was used in step 1708. A subgraph that has reached this point cannot be further reduced by the linear complexity methods of step 1708 and, further, the subgraph is not partitionable into a number of smaller subgraphs, which otherwise might be further reducible using the linear methods. So, in step 1718 the subgraph is reduced using well known more complex methods, such as, for example, the method described by Dahlhaus et al. in "The Complexity of Multiterminal Cuts", *SIAM Journal on Computing* 23, 1994, pp. 864-94. Again, in step 1710, after reducing the subgraph using these more complex methods, the subgraph is checked to determine if it is complete as a result of the reduction and, if so, returning to step 1702, where if any independent nets remain in the subgraph list, the min cut step 170 continues. Otherwise, if the subgraph is not complete, returning to step 1706 the reduced subgraph is again checked to determine if it can be further reduced using a linear complexity method.

However, if in step 1716 it is determined that more complex methods would not be effective in further reducing the subgraph, then, in step 1720, traditional or deterministic methods that may be exact or approximate, e.g. Branch & Bound, Randomized selection, etc., are applied to select an edge to be collapsed. In step 1722 the selected edge is collapsed and the modified subgraph is checked to determine if it is complete as a result of this reduction. If it is complete, returning to step 1702, if any independent nets remain in the subgraph list, the min cut step 170 continues. Otherwise, if the subgraph is not complete, returning to step 1706 the reduced subgraph is again checked to determine if it can be reduced using a linear complexity method. When in step 1702, the subgraph list is found empty, the min cut step 170 is complete in step 1710 in Figure 6A and a min cut for the communication graph has been found.

The reduction method of the preferred embodiment reduces the number of independent components in the communication graph of a complex program. In the best case, an appropriate allocation of every component in the program is provided. However, even when best case is not achieved, the preferred embodiment method may be combined with other algorithms and heuristics such as the branch and bound algorithm or the Kernighan-Lin heuristic to significantly enhance program performance. Experimentally, the present invention has been applied to communication graphs of components in several programs with results that show significant program allocation improvement, both in the quality of the final solution obtained and in the speed in reaching the result.

Although the preferred embodiments are described hereinabove with respect to distributed processing, it is intended that the present invention may be applied to any multi-task project without departing from the spirit or scope of the invention. Thus, for example, the task partitioning and distribution method of the present invention may be applied to VLSI design layout and floor planning, network reliability determination, web pages information relationship identification, and "divide and conquer" combinatorial problem solution approaches, e.g., the well known "Traveling Salesman Problem."

While the invention has been described in terms of preferred embodiments, those skilled in the art will recognize that the invention can be practiced with modification within the spirit and scope of the appended claims.